Fairness via Source Throttling:
A Configurable and High-Performance Fairness Substrate for Multi-Core Memory Systems

Eiman Ebrahimi | Chang Joo Lee | Onur Mutlu | Yale N. Patt

ASPLOS 2010 | HPS Research Group | Computer Architecture Laboratory
The University of Texas at Austin | Carnegie Mellon University
Executive Summary

- **Motivation**: Cores in a chip-multiprocessor system share multiple hardware resources in the memory subsystem
  - Interference in the shared resources can lead to unfair slowdown for some applications
- **Problem**: Existing fairness mechanisms focus on a single resource
  - Multiple independently implemented mechanisms can make contradictory decisions, leading to low fairness and loss of performance
- **Goal**: provides fairness in the entire shared memory system without degrading performance
- **Key Contributions**: Fairness via Source Throttling (FST) provides two major mechanisms
  - 1) Runtime fairness evaluation
  - 2) Dynamic request throttling
- **Major Results**: improve performance by 25.6%/14.5% and reduce unfairness by 44%/36%
Outline

- Background, Problem & Goal
- Novelty
- Key Approach & Ideas
- Mechanism
- Methodology & Evaluation
- Conclusion

- Takeaways
- Strengths and Weaknesses
- Thoughts & Ideas
- Open Discussion
Outline

- Background, Problem & Goal
- Novelty
- Key Approach & Ideas
- Mechanism
- Methodology & Evaluation
- Conclusion

- Takeaways
- Strengths and Weaknesses
- Thoughts & Ideas
- Open Discussion
System Layout
Large portion of memory subsystem is **shared** between cores.
Interference and delays lead to slowdown

Goal: all applications of equal priority experience the same slowdown
Previous Approach

- Previous research focused on individual resources
Previous Approach

Access order:
A1, A2, A3, A4, A5, A6, A7

Shared L2 cache

(a) Initial State for application A running alone

Access order:
B1, B2, B3

Shared L2 cache

(c) Initial State for application B running alone

(b) Application A’s alone memory–related stall time

(d) Application B’s alone memory–related stall time
Previous Approach

Access order:
A1, A2, A3, A4, A5, A6, A7, B1, B2, B3

Shared L2 cache

(e) Initial state for no fairness control

(f) Memory-related stall time of no fairness control

Access order:
A1, A2, A3, A4, A5, A6, A7, B1, B2, B3

Shared L2 cache

(g) Initial state for fair cache

(h) Memory-related time of fair cache
Previous research focused on individual resources.

It is challenging to properly coordinate multiple fairness mechanisms:
- Partitioning one resource may change demands on another shared resource.
Outline

- Background, Problem & Goal
- Novelty
- Key Approach & Ideas
- Mechanism
- Methodology & Evaluation
- Conclusion

- Takeaways
- Strengths and Weaknesses
- Thoughts & Ideas
- Open Discussion
System-Wide Fairness

- Tackle unfairness in the **entire shared memory system**
  - Eliminate the need for multiple fairness mechanisms

- Control fairness by **orchestrating memory requests**
  - Rate of memory request injections
  - Number of memory request injections
Outline

- Background, Problem & Goal
- Novelty
- Key Approach & Ideas
- Mechanism
- Methodology & Evaluation
- Conclusion

- Takeaways
- Strengths and Weaknesses
- Thoughts & Ideas
- Open Discussion
Interval based Estimation and Throttling

- FST works on an interval basis
- FST consists of two major components:
  1) Runtime fairness evaluation
  2) Dynamic request throttling

- No throttling in interval 1
- Throttling for Interval x is determined by the fairness estimation of interval x

Diagram:
- Interval 1: Slowdown Estimation
- Interval 2: Calculate Unfairness & Determine request rates for Interval 2 using feedback from Interval 1
- Interval 3
Outline

- Background, Problem & Goal
- Novelty
- Key Approach & Ideas
- Mechanism
- Methodology & Evaluation
- Conclusion

- Takeaways
- Strengths and Weaknesses
- Thoughts & Ideas
- Open Discussion
Runtime Fairness Evaluation

- **Goal:** dynamically estimate system unfairness

- **Slowdown:** $\frac{T_{\text{shared}}}{T_{\text{alone}}}$
  - $T_{\text{shared}}$: number of cycles to execute simultaneously with other applications
  - $T_{\text{alone}}$: number of cycles to execute alone

- **Estimating $T_{\text{alone}}$ while running multiple applications**
  - $T_{\text{excess}}$: number of excess execution cycles induced by inter-core interference
  - $T_{\text{alone}} = T_{\text{shared}} - T_{\text{excess}}$

$$IS_i = \frac{T_{\text{shared}}}{T_{\text{alone}}}, \quad \text{Unfairness} = \frac{\text{MAX}\{IS_0, IS_1, ..., IS_{N-1}\}}{\text{MIN}\{IS_0, IS_1, ..., IS_{N-1}\}}$$
Tracking Inter-Core Interference

- Three sources of inter-core interference:
  - Shared cache
  - DRAM bus and bank
  - DRAM row-buffer

- InterferencePerCore bit-vector
  - Indicate whether a core is delayed due to inter-core interference

- Bit-vector for each source
  - Update main copy by taking union of the source bit-vectors
Goal: Estimating inter-core interference on the cache by tracking cache misses caused by another core

Pollution filter for each core
- Bit-vector is indexed by the lower order bits of the accessed cache line address
- A set entry in the bit-vector indicates that a cache line belonging to this core was evicted by another core

Three steps in case of cache miss:
- 1) on cache miss access pollution filter with the missing address and check whether bit is set
- 2) set the bit in the InterferencePerCore vector and reset the bit in the pollution filter
- 3) when the interfered-with memory request is serviced reset the InterferencePerCore bit
Goal: Estimate inter-core interference caused by an inability to access DRAM due to another core using the bus or requesting service from the bank.

This situation is easily detectable:
- If detected the corresponding InterferencePerCore bit is set.

The InterferencePerCore bit is reset when no request from this core is being prevented access to DRAM by another cores requests.
Goal: Estimate interference caused by the conversion of row-buffer hits to a miss/conflict due to another cores memory request

Shadow Row-Buffer Address Register for each core and for each bank
- Whenever memory request accesses some row X, the SRAR is updated to X

Three Steps in case of Row-Buffer miss:
- 1) on row-buffer miss consult SRAR
- 2) if the SRAR bit is set, interference is present, hence InterferencePerCore bit is set
- 3) once the memory request is serviced the InterferencePerCore bit is reset
Calculate Unfairness &
Determine request rates
for Interval 2 using feedback
from Interval 1

Slowdown Estimation
Estimation of $T_{\text{excess}}$ for Core $i$

- Every cycle:
  - Check whether core $i$ experiences interference
  - Increment $T_{\text{excess}}$ by 1

- $T_{\text{alone}} = T_{\text{shared}} - T_{\text{excess}}$

\[ IS_i = \frac{T_{\text{shared}}}{T_{\text{alone}}}, \quad \text{Unfairness} = \frac{\max\{IS_0, IS_1, \ldots, IS_{N-1}\}}{\min\{IS_0, IS_1, \ldots, IS_{N-1}\}} \]
Calculate Unfairness & Determine request rates for Interval 2 using feedback from Interval 1
Dynamic Request Throttling

- Check whether the estimated unfairness is **bigger** than a certain unfairness threshold
  - Throttle **down** application with the **smallest** slowdown
  - Throttle **up** application with the **largest** slowdown

- After fairness was achieved for a certain number of successive intervals:
  - Throttle up **all** applications
Throttling Mechanisms

1) Adjust MSHR quota
   - MSHR quota determines the max. number of outstanding misses for each core
   - Reduce the pressure by decreasing the number of concurrent request contending for service

2) Adjust the rate of issuing requests to the shared cache
   - Reduce number of memory requests per unit time
   - This allows requests from other applications to be prioritized
System Software Support

- Different Fairness Objectives:
  - The goal to be achieved by FST can be configured by system software (trigger condition)

- Thread Weights:
  - Adjust priority of different applications by applying weights

- Thread Migration and Context Switches:
  - On context switch or thread migration the corresponding interference state is cleared
  - On restart of execution, the thread starts with max. throttle and then FST dynamically adapts
Scalability to more Cores

- Each core maintains a set of $N-1$ counters, with $N$ being the number of cores, which keep track of the inter-core interference caused by each other core.

- This can be used to identify which core experiences the most slowdown ($\text{App}_{\text{slow}}$) and who of the other cores is the main contributor ($\text{App}_{\text{interfering}}$).

- Once identified, the main contributor will be throttled down and $\text{App}_{\text{slow}}$ will be throttled up.

- Cores other than the $\text{App}_{\text{slow}}$ and $\text{App}_{\text{interfering}}$ are throttled up every threshold intervals to optimize performance.
FR-FCFS has the potential to **starve** application with no row-buffer locality
- Even if the interfering application gets throttled down the problem can still exist
- This denial of service can happen continuously

- Stop prioritizing row-buffer hits
Outline

- Background, Problem & Goal
- Novelty
- Key Approach & Ideas
- Mechanism
- Methodology & Evaluation
- Conclusion

- Takeaways
- Strengths and Weaknesses
- Thoughts & Ideas
- Open Discussion
System Specification

- In-house cycle-accurate x86 CMP simulator
- Faithfully model all port contention, queuing effects, bank conflicts, and other major DDR3 DRAM system constraints
Workloads

- 18 two-application workloads from the SPEC CPU 2000/2006 benchmark
  - Two-application workloads were chosen such that at least one of them is highly memory-intensive

- 10 four-application workloads from the SPEC CPU 2000/2006 benchmark
  - Four-applications workloads were chosen such that at least one of them has high intensity and one has at least medium or high intensity
Metrics

- Weighted Speedup (Wspeedup):
  - $\text{IPC}^{\text{alone}}$ is the IPC (instructions per cycle) measured when running alone.
  - $\text{IPC}^{\text{shared}}$ is measured while running in tandem with other applications.

- Harmonic mean of Speedups (Hspeedup):
  - Balanced measure between fairness and system throughput.

\[
W_{\text{speedup}} = \sum_{i=0}^{N-1} \frac{\text{IPC}^{\text{shared}}_i}{\text{IPC}^{\text{alone}}_i}
\]
\[
H_{\text{speedup}} = \frac{N}{\sum_{i=0}^{N-1} \frac{\text{IPC}^{\text{alone}}_i}{\text{IPC}^{\text{shared}}_i}}
\]
Methodology

- NoFairness:
  - Employs no fairness techniques in the shared memory subsystem
  - Uses LRU cache replacement and FR-FCFS

- FairCache:
  - Uses Virtual private caches technique for fair capacity management

- NFQ+FairCache:
  - Uses a network fair queuing (NFQ) fair memory scheduler combined with FairCache

- PAR-BS+FairCache:
  - Use parallelism-aware batch scheduling fair memory scheduler and FairCache
All Fairness techniques degrade Wspeedup to some extent

Unsophisticated fairness mechanisms can have a negative effect on system performance

FST provides a significantly better balance between system fairness and performance
Previous fairness mechanisms fail to improve system fairness significantly
  - Prioritize nonintensive applications regardless of whether or not those experience slowdown

FST is the best technique for system fairness and Hspeedup, while not falling behind in Wspeedup
Case Study

- Art and Astar are memory intensive:
  - These are slowed down too much by NFQ+FairCache and PAR-BS+FairCache, causing high unfairness
  - Inability to detect when slowdown is caused
# Hardware Cost

<table>
<thead>
<tr>
<th>Description</th>
<th>Cost for N cores</th>
<th>Cost for N = 4</th>
</tr>
</thead>
<tbody>
<tr>
<td><em>ExcessCycles</em> counters</td>
<td>( N \times N \times 16 ) bits/counter</td>
<td>256 bits</td>
</tr>
<tr>
<td>Interference pollution filter per core</td>
<td>( 2048 ) entries/( N \times (1 \text{ pollution bit} + (\log_2 N) ) bit processor id)/entry</td>
<td>24,576 bits</td>
</tr>
<tr>
<td>InterferingCoreId per MSHR entry</td>
<td>( 32 ) entries/( N \times 2 ) interference sources/( (\log_2 N) ) bits/entry</td>
<td>512 bits</td>
</tr>
<tr>
<td>InterferencePerCore bit-vector</td>
<td>( 3 ) interference sources/( N \times N \times 1 ) bit</td>
<td>48 bits</td>
</tr>
<tr>
<td>Shadow row-buffer address register</td>
<td>( N \times ) # of DRAM banks (B) \times 32 bits/address</td>
<td>1024 bits</td>
</tr>
<tr>
<td>Successive Fairness Achieved Intervals counter</td>
<td>( 2 \times N + 1 ) bits/counter</td>
<td>144 bits</td>
</tr>
<tr>
<td>Intervals To Wait To Throttle Up counter per core</td>
<td>( (2 \times N + 1) \times 16 ) bits/counter</td>
<td></td>
</tr>
<tr>
<td>Inst Count Each Interval per core</td>
<td>( 16384 ) blocks/Megabyte \times K \times (\log_2 N) bit/block</td>
<td>65,536 bits</td>
</tr>
<tr>
<td>Core id per tag store entry in K MB L2 cache</td>
<td>Sum of the above</td>
<td>92092 = 11.24 KB</td>
</tr>
<tr>
<td><strong>Total hardware cost</strong> for N-core system</td>
<td>Percentage area overhead</td>
<td>11.24KB/2048KB = 0.55%</td>
</tr>
<tr>
<td>(as fraction of the baseline K MB L2 cache)</td>
<td>Sum (KB) \times 100 / (K \times 1024)</td>
<td></td>
</tr>
</tbody>
</table>

Hardware cost of FST on a 4-core CMP system
Outline

- Background, Problem & Goal
- Novelty
- Key Approach & Ideas
- Mechanism
- Methodology & Evaluation
- Conclusion

- Takeaways
- Strengths and Weaknesses
- Thoughts & Ideas
- Open Discussion
Executive Summary & Conclusion

- **Motivation:** Cores in a chip-multiprocessor system share multiple hardware resources in the memory subsystem
  - Interference in the shared resources can lead to unfair slowdown for some applications

- **Problem:** Existing fairness mechanisms focus on a single resource
  - Multiple independently implemented mechanisms can make contradictory decisions, leading to low fairness and loss of performance

- **Goal:** provides fairness in the entire shared memory system without degrading performance

- **Key Contributions:** Fairness via Source Throttling (FST) provides two major mechanisms
  - 1) Runtime fairness evaluation
  - 2) Dynamic request throttling

- **Major Results:** improve performance by 25.6%/14.5% and reduce unfairness by 44%/36%
Outline

- Background, Problem & Goal
- Novelty
- Key Approach & Ideas
- Mechanism
- Methodology & Evaluation
- Conclusion

- Takeaways
- Strengths and Weaknesses
- Thoughts & Ideas
- Open Discussion
Takeaways

- In order to ensure good performance for multiple applications in a shared system, controlling system-wide fairness is necessary.

- By implementing FST one can decrease system complexity, due to the fact that no more coordination between multiple fairness techniques is needed.
Outline

- Background, Problem & Goal
- Novelty
- Key Approach & Ideas
- Mechanism
- Methodology & Evaluation
- Conclusion

- Takeaways
- Strengths and Weaknesses
- Thoughts & Ideas
- Open Discussion
**Strengths**

- A new approach to an old problem, which will only **get worse** with rising core counts
- In addition to improving **system-wide fairness** it also provides comparable or superior **performance** compared to prior fairness mechanisms
- Reduce system **complexity** by replacing multiple resource-based mechanisms with FST
- FST can accomplish multiple **different** fairness objectives
- The evaluation provides a good overview, while the case study provides more insight
- It is well written
Weaknesses/Limitations

- **False** positive and negative in the pollution filter
- Implementation cost of FST may **scale** poorly since the number of cores directly determines the cost
- Diminishing returns on a system with a lot of **thread migration** and **context switches**
- The optimal unfairness threshold mentioned in the paper might be hard to find
Outline

- Background, Problem & Goal
- Novelty
- Key Approach & Ideas
- Mechanism
- Methodology & Evaluation
- Conclusion

- Takeaways
- Strengths and Weaknesses
- Thoughts & Ideas
- Open Discussion
Thoughts and Ideas

- Interval-based estimation and throttling
  - What impact will an application with rapidly and randomly changing memory intensity have?

- Aggressiveness levels
  - Would it make sense to have throttle function based on slowdown instead of fixed levels?

- Security aspects are not evaluated
  - Could a single or a group of bad actors attack FST?
Outline

- Background, Problem & Goal
- Novelty
- Key Approach & Ideas
- Mechanism
- Methodology & Evaluation
- Conclusion

- Takeaways
- Strengths and Weaknesses
- Thoughts & Ideas
- Open Discussion
Discussion starters

- Will the problem become more important over time?
- Are there situations where FST might not work?
- Do you think the increase in cost due to higher bank and core counts will be overshadowed by the increase in performance?
- Can you think of some disadvantages that I missed or even some way of improving FST?
Backup Slides
Algorithm 2 Estimation of $T_{excess}$ for core $i$

Every cycle
  if inter-core cache or DRAM bus or DRAM bank or DRAM row-buffer interference then
    set InterferencePerCore bit $i$
    set InterferingCoreId in delayed memory request
  end if
  if InterferencePerCore bit $i$ is set then
    Increment ExcessCycles for core $i$
  end if

Every L2 cache fill for a miss due to interference OR
Every time a memory request which is a row-buffer miss due to interference is serviced
  reset InterferencePerCore bit of core $i$
  InterferingCoreId of core $i = i$ (no interference)

Every time a memory request is scheduled to DRAM
  if Core $i$ has no requests waiting on any bank which is busy servicing another core $j$ ($j \neq i$) then
    reset InterferencePerCore bit of core $i$
  end if
Dynamic request throttling

Algorithm 1 Dynamic Request Throttling

if Estimated Unfairness > Unfairness Threshold then
    Throttle down application with the smallest slowdown
    Throttle up application with the largest slowdown
    Reset Successive Fairness Achieved Intervals
else
    if Successive Fairness Achieved Intervals = threshold then
        Throttle all applications up
        Reset Successive Fairness Achieved Intervals
    else
        Increment Successive Fairness Achieved Intervals
    end if
end if

- This is a simplified version for dual cores
- After a certain number of fair intervals both cores are allowed to throttle up
1) Responsible for throttling down the most interfering application

2) Solving bank service denial due to FR-FCFS

3) Throttling up all applications that are neither App\textsubscript{slow} nor App\textsubscript{interfering} every threshold\textsubscript{1} intervals

4) Throttling up ??? application after number of threshold\textsubscript{2} intervals
We use 8 different aggressiveness levels:
- 2%, 3%, 4%, 5%, 10%, 25%, 50% and 100%

<table>
<thead>
<tr>
<th>Fairness Threshold</th>
<th>Successive Fairness Achieved Intervals Threshold</th>
<th>Intervals Wait To Throttle Up</th>
<th>Interval Length</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.4</td>
<td>4</td>
<td>2</td>
<td>25Kinsts</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>$\text{Switch}_{\text{thr}}$</th>
<th>$\text{Interference}_{\text{thr}}$</th>
<th>$\text{SwitchBack}_{\text{thr}}$</th>
</tr>
</thead>
<tbody>
<tr>
<td>5%</td>
<td>70%</td>
<td>3 intervals</td>
</tr>
</tbody>
</table>